Making verifiable computation a systems problem

Michael Walfish

The University of Texas at Austin
From a systems perspective, it is an exciting time for this area!

- When we started …
  - … there were no implementations
  - … my colleagues thought I was a lunatic

- Today …
  - … there is a rich design space
  - … the work can be called “systems” with a straight face
A key trade-off is performance versus expressiveness.

<table>
<thead>
<tr>
<th>setup costs</th>
<th>applicable computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>none (w/ fast worker)</td>
<td>“regular”</td>
</tr>
<tr>
<td>none</td>
<td>lower cost, less crypto</td>
</tr>
<tr>
<td>low</td>
<td>more expressive</td>
</tr>
<tr>
<td>medium</td>
<td>better crypto properties: ZK, non-interactive, etc.</td>
</tr>
<tr>
<td>high</td>
<td></td>
</tr>
</tbody>
</table>

(1) ship with compilers

(Includes only implemented systems.)
We investigate:

- What are the verifiers’ **variable** (verification, per-instance) costs, and how do they compare to native execution?

- What are the verifiers’ **fixed** (per-computation or per-batch setup) costs, and how do they amortize?

- What are the workers’ overheads?
Experimental setup and ground rules

- A system is included iff it has published experimental results.
- Data are from our re-implementations and match or exceed published results.
- All experiments are run on the same machines (2.7Ghz, 32GB RAM). Average 3 runs (experimental variation is minor).
  - For a few systems, we extrapolate from detailed microbenchmarks
- Measured systems:
  - General-purpose: IKO, Pepper, Ginger, Zaatar, Pinocchio
  - Special-purpose: CMT, Pepper-tailored, Ginger-tailored, Allspice
- Benchmarks: 150×150 matrix multiplication and clustering algorithm (others in our papers)
**Pinocchio**

setup costs are per-computation

\[
\begin{align*}
V & \quad F, EK_F \\
& \quad x^{(1)} \leftrightarrow y^{(1)} \\
& \quad x^{(2)} \leftrightarrow y^{(2)} \\
& \quad \cdots \\
& \quad x^{(B)} \leftrightarrow y^{(B)} \\
W & \quad \square
\end{align*}
\]

**Pepper, Ginger, Zaatar**

setup costs are per-batch

\[
\begin{align*}
V & \quad F, x^{(1)} \\
& \quad x^{(2)} \leftrightarrow y^{(2)} \\
& \quad \cdots \\
& \quad x^{(B)} \leftrightarrow y^{(B)} \\
W & \quad q
\end{align*}
\]
Experimental setup and ground rules

- A system is included iff it has published experimental results.

- Data are from our re-implementations and match or exceed published results.

- All experiments are run on the same machines (2.7Ghz, 32GB RAM). Average 3 runs (experimental variation is minor).
  - For a few systems, we extrapolate from detailed microbenchmarks

- Measured systems:
  - General-purpose: IKO, Pepper, Ginger, Zaatar, Pinocchio
  - Special-purpose: CMT, Pepper-tailored, Ginger-tailored, Allspice

- Benchmarks: 150×150 matrix multiplication and clustering algorithm (others in our papers)
Verification cost sometimes beats (unoptimized) native execution.

150 × 150 matrix multiplication

<table>
<thead>
<tr>
<th>Verification cost (ms of CPU time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^26</td>
</tr>
<tr>
<td>10^23</td>
</tr>
<tr>
<td>10^20</td>
</tr>
<tr>
<td>10^17</td>
</tr>
<tr>
<td>10^14</td>
</tr>
<tr>
<td>10^11</td>
</tr>
<tr>
<td>10^8</td>
</tr>
<tr>
<td>10^5</td>
</tr>
<tr>
<td>10^2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

- Pepper
- Ginger
- Zaatar
- Pinocchio

Ishai et al. (PCP-based efficient argument)

50 ms

5 ms
Some of the general-purpose protocols have reasonable cross-over points.

- Ginger (slope: 18 ms/inst) cross-over point: 4.5 million instances
- Pinocchio (slope: 12 ms/inst)
- Zaatar (slope: 33 ms/inst)
- Native (slope: 50 ms/inst)

Verification cost (minutes of CPU time) vs. instances of 150x150 matrix multiplication.
The cross-over points can sometimes improve with special-purpose protocols.
The worker’s costs are pretty much preposterous.
Summary of performance in this area

- None of the systems is at true practicality
- Worker’s costs still a disaster (though lots of progress)
- Verifier gets close to practicality, with special-purposeness
  - Otherwise, there are setup costs that must be amortized
  - (We focused on CPU; network costs are similar.)
<table>
<thead>
<tr>
<th>setup costs</th>
<th>“regular”</th>
<th>straightline</th>
<th>pure, no RAM</th>
<th>stateful, RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Thaler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w/ fast worker)</td>
<td>[CRYPTO13]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>CMT</td>
<td>Allspice</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ITCS12]</td>
<td>[Oakland13]</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>Pepper</td>
<td>Ginger</td>
<td>Zaatar</td>
<td>Pantry</td>
</tr>
<tr>
<td></td>
<td>[NDSS12]</td>
<td>[Security12]</td>
<td>[Eurosys13]</td>
<td>[SOSP13]</td>
</tr>
<tr>
<td>medium</td>
<td></td>
<td></td>
<td>Pinocchio</td>
<td>Pantry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Oakland13]</td>
<td>[SOSP13]</td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pantry [SOSP13] creates verifiability for real-world computations

before:

- V supplies all inputs
- F is pure (no side effects)
- All outputs are shipped back

after:

query, digest

result

map(), reduce(),
input filenames

output filenames
Recall the compiler pipeline.

(The last step differs among Ginger, Zaatar, Pinocchio.)
Programs compile to constraints on circuit execution.

```
dec-by-three.c

f(X) {
    Y = X - 3;
    return Y;
}
```

Input/output pair correct $\iff$ constraints satisfiable.

As an example, suppose $X = 7$.

- If $Y = 4$ ...
  \[
  \begin{cases}
    0 = Z - 7 \\
    0 = Z - 3 - 4
  \end{cases}
  \]
  ... there is a solution

- If $Y = 5$ ...
  \[
  \begin{cases}
    0 = Z - 7 \\
    0 = Z - 3 - 5
  \end{cases}
  \]
  ... there is no solution
The pipeline decomposes into two phases.

```
F() {
  [subset of C]
}
```

\[
\begin{align*}
0 &= X + Z_1 \\
0 &= Y + Z_2 \\
0 &= Z_1Z_3 - Z_2 \\
\ldots
\end{align*}
\]

constraints (\(E\))

```
V
```

```
W
```

If \(E(X=x, Y=y)\) is satisfiable, computation is done right.

Design question: what can we put in the constraints so that satisfiability implies correct storage interaction?
How can we represent storage operations? (1)

Representing “load(addr)” explicitly would be horrifically expensive.

Straw man: variables $M_0, \ldots, M_{\text{size}}$ contain state of memory.

$B = \text{load}(A)$

\[
\begin{align*}
B &= M_0 + (A - 0) \cdot F_0 \\
B &= M_1 + (A - 1) \cdot F_1 \\
B &= M_2 + (A - 2) \cdot F_2 \\
&\quad \ldots \\
B &= M_{\text{size}} + (A - \text{size}) \cdot F_{\text{size}}
\end{align*}
\]

Requires two variables for every possible memory address!
How can we represent storage operations? (2)

Consider **self-certifying blocks:**

- They bind references to values
- They provide a substrate for verifiable RAM, file systems, …

  [Merkle CRYPTO87, Fu et al. OSDI00, Mazières & Shasha PODC02, Li et al. OSDI04]

**Key idea:** encode the hash checks in constraints

- This can be done (reasonably) efficiently

Folklore: “this should be doable.” (Pantry’s contribution: “it is.”)
We augment the subset of C with the semantics of untrusted storage

- block = vget(digest): retrieves block that must hash to digest
- hash(block) = vput(block): stores block; names it with its hash

```
add_indirect(digest d, value x) {
  value z = vget(d);
  y = z + x;
  return y;
}
```

Worker is obliged to supply the “correct” Z (meaning something that hashes to d).
Putting the pieces together

- recall: \( \square = \text{“I know a satisfying assignment to } E(X=x, Y=y) \text{”} \)
- checks-of-hashes pass \( \iff \) satisfying assignment identified
- checks-of-hashes pass \( \iff \) storage interaction is correct
- storage abstractions can be built from \{vput(), vget()\}
The verifier is assured that a MapReduce job was performed correctly—without ever touching the data.

The two phases are handled separately:

Mappers:
- \( \text{in} = \text{vget(in\_digest)} \)
- \( \text{out} = \text{map(in)} \)
- for \( r=1,\ldots,R \):
  - \( d[r] = \text{vput(out[r])} \)

Reducers:
- for \( m=1,\ldots,M \):
  - \( \text{in}[m] = \text{vget(e[m])} \)
  - \( \text{out} = \text{reduce(in)} \)
  - \( \text{out\_digest} = \text{vput(out)} \)
Example: for a DNA subsequence search, the verifier saves work, relative to performing the computation locally.

- A mapper gets 1000 nucleotides and outputs matching locations
- Vary mappers from 200 to 1200; reducers from 20 to 120
Pantry applies fairly widely

- Our implemented applications include:
  - Verifiable queries in (highly restricted) subset of SQL
  - Privacy-preserving facial recognition
  - Our implementation works with Zaatar and Pinocchio
Major problems remain for this area

- Setup costs are high (for the general-purpose systems)

- Verification costs are high, relative to native execution
  - Evaluation baselines are highly optimistic
  - Example: $100 \times 100$ matrix multiplication takes 2 ms on modern hardware; no VC system beats this.

- Worker overhead is $1000 \times$

- The computational model is a toy
  - Loops are unrolled, memory operations are expensive
Summary and take-aways

- A framework for organizing the research in this area is performance versus expressiveness.

- Pantry extends verifiability to stateful computations, including MapReduce, DB queries, RAM, etc.

- Major problems remain for all of the systems
  - Setup costs are high (for the general-purpose systems), and verification does not beat optimized native execution
  - Worker costs are too high, by many orders of magnitude
  - The computational model is a toy